Toward a Unified Approach to Coupling

Aaron B. Binkley and Stephen R. Schach
Computer Science Department, Vanderbilt University, Nashville, TN
{binkley, srs}@vuse.vanderbilt.edu

Abstract

We describe a unified approach to coupling that incorporates four disparate elements, namely, taxonomies for classical coupling; taxonomies for object-oriented coupling; metrics for classical coupling; and metrics for object-oriented coupling. A single instance of coupling between two modules (or classes) can incorporate several points of dependency between the two modules (or classes). It is the combination of the effects of these dependencies that gives the true measure of that instance of coupling. We identify a metric based on three types of dependencies that commonly exist between modules, namely, *referential dependency*, a measure of the extent to which the program relies on its declarations remaining unchanged; *structural dependency*, a measure of the extent to which the program relies upon its internal organization remaining unchanged; and *data integrity dependency*, a measure of the vulnerability of data elements in one module to change by other modules. We show how this approach can be used to describe different forms of coupling. We also compare our metric with another coupling metrics.

1. Introduction

The coupling between two classes is a measure of the degree of dependency between the classes [10]. High coupling has a deleterious effect on both maintainability and reusability [12]. Accordingly, measuring the coupling between pairs of classes is an important design metric.

If we wish to measure the coupling between two classes, there are four different approaches that we can adopt. First, in view of the fact that a class is a special case of a classical module, we can use one of the many taxonomies of the classical (structured) paradigm, such as that of [10] or [12]. Thus, the classical coupling between two classes can be described as, say, common coupling or data coupling. Second, we can classify the coupling according to one of the many taxonomies specific to the object-oriented paradigm, including [2] or [13]. Using such a taxonomy, coupling between two classes might be classified as, say, interface coupling or outside internal object coupling from the side.

Notwithstanding the widespread use of classical and object-oriented coupling taxonomies, these approaches have an inherent problem, namely, the level of granularity at which they are applied. A major purpose of evaluating coupling is to determine maintainability and reusability. These two qualities are affected by a variety of factors. For example, when the coupling is via a calling interface, the factors include the number of parameters and whether they are passed by value or by reference. Even though the many published taxonomies differ as to details, all agree that class A passing a simple data item to class B (data coupling) is preferable to the situation where classes A and B can both potentially modify the same global variable (common coupling). However, what if class A passes 57 simple data items to class B? In terms of the standard taxonomies this is indeed data coupling, but both maintainability and reusability are adversely affected. In fact, most designers would prefer to have common coupling with one shared variable than to have data coupling with an excessive number of parameters. Another problem with taxonomies is that they are ordered, not numeric. That is, we can say that coupling category C_1 is better than category C_2, but we cannot attach a numerical value to either of the two categories as we can with a less qualitative metric. Because of these problems, we believe that existing coupling taxonomies are inadequate from the viewpoint of measuring the coupling between two classes.

The other two approaches we can take to measuring the coupling between two classes are classical coupling metrics and object-oriented coupling metrics. There are a variety of classical coupling metrics, including metrics based on fan-in and fan-out [8] and on measures of complexity [5, 7]. Again, we believe that these metrics are at too high a level of granularity. For example, fan-out is the same in two different instances of common coupling, one in which two classes access a global data element by name, the other in which they do so by giving a byte offset into a shared memory area. The two situations are different from the viewpoints of maintainability and reusability, and this should be reflected in the coupling metric.

Instead of applying classical metrics to the object-oriented domain, specifically object-oriented metrics can be used. Some of these are applicable to coupling, such as the metrics of [6] and [9]. However, here again the level of granularity does not permit the designer to distinguish between variations of the forms of coupling measured by those metrics. For example, the value of the coupling between object classes (CBOC) metric [6] is the same for a given class, regardless of the number of interface items from other classes it actually references.

In addition to the difficulties specific to each approach, there is one problem that is common to all four, namely, the fact that there are four different approaches. What is needed is a unifying approach to measuring coupling that is equally applicable to both paradigms.

In this paper, we describe the coupling dependency metric (or CDM), a coupling metric that solves these problems. First, the level of granularity is low enough to distinguish between subclasses of coupling. Second, the metric is domain-independent; it is equally applicable to the structured and object-oriented domains.

The basis for our metric is that a single instance of coupling between two modules (or classes) can incorporate several points of dependency between the two modules; it is the combination of the effects of these dependencies that gives the true measure of that instance of coupling. We identify three types of dependencies that commonly exist between modules. *Referential dependency* is a measure of the extent to which the program relies on its declarations remaining unchanged; *structural dependency* measures the extent to which the program relies upon its internal organization remaining unchanged; *data integrity dependency* is a measure of the vulnerability of data elements in one module to change by other modules. We obtain an overall measure of the coupling by combining these three dimensions of coupling.

In the next section, we define the coupling dependency metric. In Section 3 we calculate it for declaration coupling, a recently identified form of coupling. The CDM for classical forms of coupling is presented in Section 4, and for object-oriented forms in Section 5. Comparisons between our metric and other coupling metrics are presented in Section 6. A discussion and conclusions are presented in Section 7.

2. Defining the Coupling Dependency Metric

Coupling exists between two modules (or classes) when a change made to one module (or class) can potentially change the other module (or class). Instead of attempting to use just one dimension to measure coupling, we propose three distinct dependency dimensions (or *facets*). In this section we define the three constituent facets of the coupling dependency metric.

The dependency between two modules (or classes) is then measured with respect to each of these three facets in *dependency units* (DU), a discrete integer scale. The resulting dependency vector is defined to be the *coupling dependency metric* (CDM). This is a measure of the coupling between the respective two modules (or classes). The facets can be used separately, or they can be summed to provide a single composite measure of the instance of coupling.

We now define our three dimensions. The first is *referential dependency*, a measure of the extent to which a program relies on its declarations remaining unchanged. If a module A makes a reference to a component element of a module B (either explicitly or implicitly), this makes module A dependent on the stability of module B. Thus, each reference within module A to a distinct component element in another module introduces one unit of dependency. Each such reference is represented by the dependency vector R = (1, 0, 0). For example, if module A accesses a variable x declared within module B by giving the name of the variable, module A is referentially dependent on module B.

The second of the dimensions is that of *structural dependency*, and it measures the extent to which a program relies on its internal organization remaining unchanged. The sharing of information between modules written in a high-level language is often constrained by the physical structure of the design of the program, often hierarchical in the case of the object-oriented paradigm. It is important to measure the extent to which modules depend on their relative positions in the overall program structure. Each such constraint on the program structure introduces a unit of dependency and each can be represented by the dependency vector S = (0, 1, 0). For example, in a language such as Pascal, if module A calls module B, module B must have been defined before A in the program, and this imposes a structural dependency between the two modules.

The third and last of the dimensions is that of *data integrity dependency*, a measure of the vulnerability of data elements in one module to change by other modules. When module A shares information with module B, it is important to measure the level of vulnerability introduced to the program because of this sharing. One unit of dependency is introduced for each data element within module A for every other module that may alter the value of that element, and each is represented by the dependency vector D = (0, 0, 1). For example, if module B may change the value of variable x which is declared within module A, the data integrity of A is dependent on module B.

Next, all the individual R, S, and D vectors are summed. The value of the CDM metric is then the resulting vector. The component values of the vector can be used separately (e.g., to express the total structural dependency in an instance of coupling), or the facets can be summed to provide a single composite measure of the dependencies within the instance of coupling.

To illustrate how this works, we now show how these three facets occur in a well-known form of classical coupling, namely, common coupling. Common coupling occurs when two modules can both potentially modify the same variable [10, 12]. When a data element is shared between two modules through common coupling, there exist several dependencies which can be measured in their appropriate dimensions. In the Pascal* code of Figure 1, for example, data element x of **procedure** p1 is accessed by **procedure** p3, and the following dependencies exist in this typical example of common coupling:

1. Procedure p3 is dependent on the declaration of the name x in p1; that is, p1 must not change the name of variable x because of the reference to it in p3 (1 DU of referential dependency).

* One might wonder why Pascal was chosen instead of a language more commonly used in industry (e.g., COBOL or C), but a hierarchical language providing the nesting of modules was needed for this example.

```
procedure p1;
        var x : integer;
    procedure p3;
    begin (* p3 *)
        for x := 0 to 100 do
            writeln (x)
    end;
begin (* p1 *)
    p3
end;
```

Figure 1: Simple example of common coupling.

2. Procedure p3 is dependent on the type declaration of x in p1; that is, p1 must not change the type of variable x because of the assumption within p3 that x is an integer. This is an implicit reference by p3 to the type of x (1 DU of referential dependency).
3. Because of the rules for the scope of identifiers in Pascal, p3 must be nested within p1 in order to have access to the variables of p1. Therefore, the accessing of x is dependent on the structure p1 ⊃ p3 [read: "p1 contains p3"] (1 DU of structural dependency).
4. Procedure p3 can change the value of x (1 DU of data integrity dependency).

Combining the above component dependencies, the dependency vector for this simple instance of common coupling is (2, 1, 1).

3. Calculating CDM for Declaration Coupling

Before we can consider other forms of coupling, both classical and object-oriented, we need to consider a recently identified form of coupling, namely, *declaration coupling* [3]. Although a distinct form of coupling, it usually occurs in combination with other categories of coupling. Declaration coupling is treated separately in this section so that once its effects on maintenance and reuse are quantified, these results can be combined with additional dependencies to express more succinctly the complete effects of the other forms of coupling which are encountered in the classical and object-oriented paradigms.

Declaration coupling is a form of coupling found in both the classical and object-oriented paradigms. It is defined as follows: Let A and B be mutually disjoint modules. If module A contains a declaration (that is, a definition of the implementation of a data structure or a code sequence, or both) and if there is an (implicit or explicit) instance of that declaration in module B, then A and B are *declaration coupled*. (As will be described later, the phrase "contains a declaration" includes an implicit null declaration.)

In the Pascal code of Figure 2, for example, procedure p3 is declaration coupled to procedure p1 because p3 uses a data type that was defined in p1. This induces the following dependencies:

```
procedure p1;
    type percent = 0 .. 100;
    procedure p3;
        var x : percent;
    begin (* p3 *)
        for x := 0 to 100 do
            writeln (x)
    end;
begin (* p1 *)
    p3
end;
```

Figure 2: Simple example of declaration coupling.

```
procedure p1;
        type percent = 0 .. 100;
    procedure p2;
        procedure p3;
            var x : percent;
        begin (* p3 *)
            for x := 0 to 100 do
                writeln (x)
        end;
    begin (* p2 *)
        p3
    end;
begin (* p1 *)
    p2
end;
```

Figure 3: More complex declaration coupling

1. Procedure p3 is dependent on the declaration of the name percent in p1; that is, p1 must not change the name of type percent because of the reference to it in p1 (1 DU of referential dependency).
2. Procedure p3 is dependent on the type declaration of percent in p1; that is, p1 must not change the declaration of percent because of the reference in p3 to the physical attributes of percent data elements. This is an implicit reference by p3 to the declaration of percent (1 DU of referential dependency).
3. Because of the rules for the scope of identifiers in Pascal, p3 must be nested within p1 in order to have access to the type declarations of p1. Therefore, the use of percent is dependent on the structure p1 ⊃ p3 (1 DU of structural dependency).

Given the above dependencies, the vector for this simple instance of declaration coupling is therefore (2, 1, 0).

Declaration coupling between modules can take a more complicated form as the Pascal example of Figure 3 illustrates. In this example, p3 is still sharing percent with p1, but the further condition exists that p3 is now nested within procedure p2. At first glance, one might say that p2 is independent of the declaration coupling of modules p1 and p3, but that is not the case. If a new identifier percent were declared within p2, the reference to percent in p3 would no longer be bound to the declaration of percent within p1, but instead it would be bound to the declaration of percent within p2. Because we cannot make arbitrary changes to p2 without considering the implications they may have on the declaration coupling of p1 and p3, p2 is not, in fact, independent of p1 and p3 with regard to the sharing of type percent. Therefore, we must measure this dependency as well. Along with the previous dependencies, there is a fourth dependency here, namely:

4. Procedure p3 is dependent on an implicit null declaration of percent in p2; that is, p2 must not declare a local identifier percent. This is an instance of declaration coupling between p2 and p3 (1 DU of referential dependency).

Although the above dependency is between p2 and p3, we include its negative effects into the measure of the declaration coupling between p1 and p3 because the coupling exists as a consequence of the declaration coupling between p1 and p3.

Each instance of implicit null declaration coupling introduces one unit of referential dependency into the program (due to the dependency on the null declaration) and is described using the dependency vector (1, 0, 0). There will be one such unit for each level of nesting.

The overall dependency vector for the instance of declaration coupling between p1 and p3 shown in Figure 3 is thus (3, 1, 0). In general, the declaration coupling between two modules is described by the vector $[(2, 0, 0) + \Sigma D + \Sigma S]$, where ΣD reflects the dependencies introduced by implicit null declaration coupling and ΣS reflects the dependencies introduced by the structural restrictions imposed by the programming language. In Pascal, for example, structural dependency is induced by the need for nesting in situations such as that depicted in Figure 3.

4. Calculating CDM for Classical Forms of Coupling

As previously defined, the term *classical coupling* refers to the taxonomy of coupling first defined by Stevens, Myers, and Constantine in their landmark 1974 paper [12]. In the previous section, we showed how to derive CDM for a simple form of common coupling. We have also derived CDM for the classical types of coupling. For the sake of brevity, we omit the details and give only the results here. In all five formulae that follow, ΣD reflects the dependencies introduced by the declaration couplings present (including implicit null declaration couplings), and ΣS reflects the dependencies introduced by the structural restrictions imposed by the programming language.

Content Coupling	$[(2, 0, \lvert CS\rvert) + \Sigma D + \Sigma S]$, where $\lvert CS\rvert$ is the number of instructions in the code segment being accessed.
Common coupling	$[(2, 0, v) + \Sigma D + \Sigma S]$, where v is the total number of variables shared between the two modules.
Control Coupling	$[(2Pt + \lvert CV\rvert + 2, 0, 0) + \Sigma D + \Sigma S]$, where Pt is the total number of parameters, and $\lvert CV\rvert$ is the number of possible values that the control variable can assume.
Stamp Coupling	$[(2Pt + 2, 0, Ar) + \Sigma D + \Sigma S]$, where Pt is the total number of parameters, and Ar is the total number of atomic data elements passed between the two modules by reference (as opposed to simply the number of parameters passed by reference).
Data Coupling	$[(2Pt + 2, 0, Pr) + \Sigma D + \Sigma S]$, where Pt is the total number of parameters, and Pr is the total number of parameters passed by reference.

```
class Quadrilateral : public Polygon {
    private:
        float area;
    public:
        void setarea (float newarea);
};
void Quadrilateral :: setarea (float newarea) {
    area = newarea;
}
```

Figure 4: Sample C++ class declaration.

5. Calculating CDM in the Object-Oriented Paradigm

We turn now to some published taxonomies of object-oriented coupling categories. Again, for the sake of brevity, we explicitly show the derivation for just one type of coupling, and then cite the results for others.

If object A references a component of the public interface of object B, objects A and B are *interface coupled* [13]. This usually takes the form of object A invoking a method of object B or changing the value of an instance variable of object B. For example, given the C++ code of Figure 4, if some object A invokes the setarea method of an object B of class Quadrilateral, this induces the following dependencies:

1. Object A is dependent on the declaration of method setarea; that is, method setarea must not be renamed because of the reference to it in object A (1 DU of referential dependency).
2. Object A is dependent on the type declaration of the return value of method setarea; that is, method setarea must not return a result which is incompatible with what object A expects (1 DU of referential dependency).

```
class Polygon : public Shape {
    private:
        float area;
    public:
        void setarea (float newarea);
};

class Quadrilateral : public Polygon {
};

void Polygon :: setarea (float newarea) {
    area = newarea;
}
```

Figure 5: C++ Class inheriting a method.

3. Object A is dependent on the type declaration of parameter newarea of method setarea. This is an implicit reference by object A to the type of newarea (1 DU of referential dependency).
4. Because of the rules for the scope of identifiers in C++, the definition of Polygon must precede the definition of Quadrilateral, that is, Polygon ⊃ Quadrilateral (1 DU of structural dependency).

Given the above component dependencies, the dependency vector for this typical instance of interface coupling is (3, 0, 1).

Additional dependencies are introduced, though, when methods are passed multiple parameters. In such a case, the invoking object must match its parameter list to that of the method it invokes, and this requirement induces the following dependencies:

5. In C++, formal and actual parameters are paired according to their relative positions in the parameter lists. This assigns each formal parameter a unique name corresponding to its ordinal position in the list. Therefore, the invoking method is referentially dependent on the ordinal names of the parameters of the method (1 DU of referential dependency per parameter).
6. The invoking object is also dependent on the type declarations of the additional parameters of the method (1 DU of referential dependency per parameter).

In addition, there are three other situations which may complicate the interface coupling between two objects, namely, when the method being invoked is an inherited method, when the inheritance of the method is through multiple generations, and when there are additional attributes listed in the public interface.

The C++ code of Figure 5 shows an example of the first complication. Class Quadrilateral inherits method setarea from class Polygon. If an object invokes the setarea method of an object of class Quadrilateral, the object is dependent on the presence of setarea in the public interface of class Quadrilateral, and by transitivity the following dependencies are induced:

```
class Polygon : public Shape {
    private:
        float area;
    public:
        void setarea (float newarea);
};

class Quadrilateral : public Polygon {
};

class Square : public Quadrilateral {
};

void Polygon :: setarea (float newarea) {
    Polygon :: area = newarea;
}
```

Figure 6: Inheritance through multiple generations.

7. The interface of Quadrilateral is dependent on that class's inheritance from Polygon; that is, class Quadrilateral may not be moved in the class hierarchy so that it is no longer a descendant of Polygon. Therefore, the interface of the Quadrilateral object is dependent on the structure Polygon ← Quadrilateral (read: "Polygon is an ancestor of Quadrilateral") (1 DU of referential dependency).
8. The interface of Quadrilateral is dependent on the definition of the setarea method of class Polygon being public. If its definition were later changed to private, Quadrilateral could no longer inherit it. Therefore, there is an implicit reference to the definition of the access type of setarea (1 DU of referential dependency).

The situation is further complicated when, as in Figure 6, the second condition exists, that is, the inheritance of the method is through multiple generations. Assuming that an object invokes the setarea method (inherited from Polygon) of a Square object, the following additional dependencies are induced:

9. Not only is the interface of class Square dependent on Quadrilateral ← Square, but also on Polygon ← Quadrilateral. In general, there will be additional referential dependencies equal to the length of the path between the descendant class and the ancestor class from which it inherits a method.
10. The interface of Square is dependent on an implicit null declaration of method setarea within class Quadrilateral. If Quadrilateral were to have a method defined with that name, Square would inherit the method from Quadrilateral instead of from Polygon, and this might cause a number of undesirable effects. In general, there

```
class Quadrilateral : public Polygon {
    public:
        float area;
        char name [15];
        void setarea (float newarea);
};

void Quadrilateral :: setarea (float newarea) {
    area = newarea;
}
```

Figure 7: Sample C ++ public interface.

is implicit null declaration coupling between a class and each of its ancestor classes along the path between the descendant class and the ancestor class from which it inherits a method.

Finally, Figure 7 shows an example of the third complication, namely, there are additional attributes listed in the public interface. If the designers of a class include attributes in the public interface of the class, this has the undesirable effect of giving access to all of these attributes to those modules which reference objects of that class. This induces a unit of data integrity dependency for each attribute listed in the public interface.

Combining all these component dependencies, the dependency vector for a typical instance of interface coupling is $[(2Pt + 3, 0, v) + \Sigma D + \Sigma S]$, where Pt is the total number of parameters, v is the number of variables in the public interface, ΣD reflects the dependencies introduced by the declaration couplings present (including implicit null declarations), and ΣS reflects the dependencies introduced by the structural restrictions imposed by inheritance.

Again, for brevity, we simply cite the coupling dependency metric for the object-oriented coupling categories listed in [2]. In all four formulas that follow, ΣD reflects the dependencies introduced by the declaration couplings present and ΣS reflects the dependencies introduced by the structural restrictions imposed by the programming language.

Interface Coupling	$[(2Pt + 3, 0, v) + \Sigma D + \Sigma S]$ where Pt is the total number of parameters and v is the number of variables in the public interface.
Outside Internal Object Coupling from the Side	$[(3, 0, v) + \Sigma D + \Sigma S]$, where v is the total number of variables (or attributes) in the accessed class.
Outside Internal Object Coupling from Underneath	$[(3, 0, v) + \Sigma D + \Sigma S]$, where v is the total number of variables (or attributes) visible from underneath the accessed class.
Inside Internal Object Coupling	$[(2, 0, v) + \Sigma D + \Sigma S]$, where v is the total number of variables (or attributes) in the container class.

A useful side effect of CDM is that it assigns a "signature" (*rd*, *sd*, *di*) to an instance of coupling. First, the signature of common coupling is identical to that of inside internal object coupling. This is in accord with the claim that every category of object-oriented coupling reduces to a category of classical coupling, and that inside internal object coupling is simply another name for common coupling [11]. Second, the signatures of outside internal object coupling from the side and outside internal object coupling from underneath are the same. This implies that the two categories of object-oriented coupling have similar negative effects on reusability and maintainability.

6. Comparison with Other Metrics

We compared the five CDM formulae for classical coupling presented in Section 4 with classical coupling taxonomies. We computed the composite value of CDM for a large range of values of the parameters. In almost all cases, the resulting ordering of the value of CDM for the five formulae is precisely that of the corresponding five classical coupling categories in the taxonomies of [10] and [12]. That is, the worse the coupling, the higher the value of CDM. The CDM metric is therefore plausible, at least within the classical domain.

With regard to the object-oriented domain, we tested CDM using data presented in [1]. The authors of that paper submitted two or three versions of various object-oriented design fragments to experts and asked those experts to decide which version had the better design. They then compared the experts' opinion with the result of applying their permitted interaction metric (PIM) to the same versions. When we applied our CDM to the 18 design fragments in [1], we found that in 14 out of the 18 cases (78%) CDM agreed with the experts as to which was the better design; these results are fully discussed in [4]. Thus, CDM appears to perform well as a measure of object-oriented design quality.

7. Discussion and Conclusions

We have developed the coupling dependency metric (CDM), a metric for coupling that incorporates three different dimensions or facets. The CDM between two modules is the 3-dimensional vector representing the three facets of coupling. The facets can be used separately, or they can be summed to provide a composite measure of the dependencies in an instance of coupling.

One strength of CDM is its multidimensional (multifaceted) nature. A problem with almost all previous coupling metrics is that they essentially measure a single quality, such as fan-in/fan-out or a count of the number of other classes to which a given class is coupled. In contrast, CDM measures three distinct quantities, namely, *referential dependency*, a measure of the extent to which the program relies on its declarations remaining unchanged; *structural dependency*, a measure of the extent to which the program relies upon its internal organization remaining unchanged; and *data integrity dependency*, a measure of the vulnerability of data elements in one module to change by other modules. Taking the sum of these three dimensions of coupling we obtain an overall measure of an instance of coupling. In this way, we can incorporate what we believe to be all the facets of coupling into one number.

In our opinion, however, the greatest strength of CDM is that is unifies the classical and object-oriented paradigms. That is, instead of having two different taxonomies or two different sets of metrics, one for software developed using the classical paradigm and the other for software developed using the object-oriented paradigm, there is just one metric applicable to both paradigms that replaces both the two taxonomies and the two sets of metrics.

References

[1] D. H. Abbott, T. D. Korson, and J. D. McGregor, "A Proposed Design Complexity Metric for Object-Oriented Development," Technical Report 94–105, Clemson University, Clemson, SC, April 1994.

[2] E. Berard, *Essays in Object-Oriented Software Engineering*, Prentice-Hall, Englewood Cliffs, NJ, 1993, pp. 72–130.

[3] A. B. Binkley and S. R. Schach, "A Classical View of Object-Oriented Cohesion and Coupling," Technical Report 96–06, Computer Science Department, Vanderbilt University, Nashville, TN, 1996.

[4] A. B. Binkley and S. R. Schach, "A Comparison of Sixteen Quality Metrics for Object-Oriented Design," *Information Processing Letters* **57** (No. 6, 1996), pp. 271–275.

[5] D. N. Card and R. L. Glass, *Measuring Software Design Quality*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[6] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering* **20** (June 1994), pp. 476–493.

[7] B. Henderson-Sellers and J. M. Edwards, *The Working Object*, Prentice-Hall, Englewood Cliffs, NJ, 1994.

[8] S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering* **7** (May 1981), pp. 510–518.

[9] M. Lorenz, *Object-Oriented Software Development: A Practical Guide*, Prentice-Hall, Englewood Cliffs, NJ, 1993, p. 227.

[10] S. R. Schach, *Classical and Object-Oriented Software Engineering*, Third Edition, Richard D. Irwin, Chicago, 1996.

[11] S. R. Schach, "On the Cohesion and Coupling of Objects: A Classical Approach," *Journal of Object-Oriented Programming* **8** (January 1996), PP. 48–50.

[12] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," *IBM Systems Journal* **13** (No. 2, 1974), pp. 115–139.

[13] F. H. Wild, "Managing Class Coupling," *UNIX Review* **9** (October 1991), pp. 44–47.